The Craft of API Design

Clifford Wolf

ROCK Linux - http://www.rocklinux.org/

Csync2 - http://oss.linbit.com/csync2/

STFL - http://www.clifford.at/stfl/

SPL - http://www.clifford.at/spl/

Introduction

What are APIs? (1/2)

- APIs are the (programatical) face of any piece of programming work.
- This is not limited to libraries intended for a broad use.
- It also covers the internal APIs found in every non-trivial program.

What are APIs? (2/2)

- APIs can be seen as extensions to the functionality of a programming language.
- A programming language comes with a very limited built-in vocabulary (set of functions and keywords).
- Every program module extends this vocabulary using its API.

Why are APIs so important?

- Today's programming projects tend to become huge pretty fast.
- It is hard to fully understand a huge program.
- So projects are broken up into modules with the APIs connecting them.
- With this seperation, it enables the programmer to focus on one module without the need of knowing the rest of the project inside out.
- This can't work out unless the APIs are stable and cleanly seperating the modules from each other.

Why is API-Design so important?

- Projects tend to “grow” with the APIs happening instead of beeing designed.
- A well designed API enables the programmer to change a module’s internals without the need for changing the API or other modules.
- So if the API is good, everything else can also be improved later in the process.
- But changing an API is always complicated, especially if the API has grown already a big user base.

APIs Everywhere

Frameworks

- Frameworks are getting more important than the programming languages they are written for and in.
- Example given:
  - Ruby on Rails

Standard APIs (1/2)

- Some APIs are part of standards and independend of implementations.
- Example given:
  - Standard C Library
  - The POSIX system call layer
  - The Verilog PLI layer
  - The MPI API

- A thing to ponder:
  Usually it takes a few hours to learn a new language, learning the standard libraries API is the hard part.

Standard APIs (2/2)

- Some APIs are even independend of programming languages.
- Example given:
  - The DOM API

- Changing a standard API once it is published and in use is close to impossible.

Toolchain Libraries

- Most of today's applications don't implement a single non-trivial algorithm.
- Instead ready-to-use libraries are used.
- Example given:
  - APIs to sort functions vs. sorting algorithms - Which one is more important for the daily work of an application programmer?

Program Modules

- Today's software projects are usually split up in small modules.
- This is pimarily done to fight complexity.
- The whole effort is useless if the APIs between this modules aren't well designed.

Some Guidelines

Introduction (1/2)

- I will present some Guidelines for API design.
- None of them is an absolute rule.
- But I believe they bring up some questions everyone designing an API should worry about.
- Feedback is always welcome.

Introduction (2/2)

- The following rules and things to ponder might be useful:
  - For designing new APIs
  - For cleaning up existing APIs
  - For judging others APIs

- Every minute cut in API design **will** hit you hard later.

Hard to use wrong (1/2)

- An API must be hard to use wrong.
- This is different from easy to use right.

Hard to use wrong (2/2)

Good example:

```
char *get_current_dir_name(void);
```

Returns a **malloced** string containing the absolute pathname of the current working directory. (GNU extension)

Bad example:

```
char *strncpy(char *dest, const char *src, size_t n)
```

If there is no null byte among the first n bytes of src, the result will not be null-terminated.

Be self-descriptive (1/2)

- The name should indicate what it does.
- Use names for function parameters when the language provides them.
- The API should pass the "Telephone test".

Good example:

```
egrep --recursive --exclude-dir .svn Copyright .
```

Bad example:

```
char *exclude_dir_vect[] = { ".svn", NULL };
fictional_egrep_function(0, 0, 1, 0, 0, NULL, NULL,
    exclude_dir_vect, NULL, NULL, "Copyright", ".");
```

Be self-descriptive (2/2)

Good example:

```
if (strstr(haystack, "needle"))
    printf("Found a needle!\n");
```

Bad example:

```
print "Found a needle!\n" if haystack =~ /needle/;
```

Good example:

```
int on_exit(void (*function)(int , void *), void *ar
```

Bad example:

```
sighandler_t signal(int signum, sighandler_t handler
```

Hide something

- An API must hide something.
- The inner workings of a module must not dictate the API.

Good and bad example:

```
void qsort(void *base, size_t nmemb, size_t size,
    int(*compar)(const void *, const void *));
```

Don’t overoptimize

- Allow for changes in the algorithm.
- This is where optimization happens!

Common failure: Trading a ’fast API’ for less flexibility.

Bad Example:

```
void qsort(void *base, size_t nmemb, size_t size,
    int(*compar)(const void *, const void *));
```

(A user context pointer should be passed thru to compare.)

Support User Contexts

- Always pass a void user context pointer to callback functions.
- Global variables should never be used for this!

- Always configure a library using some kind of context object.
- Global variables should never be used for this!

- Bad examples: qsort(), LibXML2
- Good examples: epool, PCRE

Be Pragmatic

- Each module/function should do one thing well.

Bad examples:

- Using select() as nanosleep/usleep replacement. (4.2BSD)

Good examples:

- BSD socket API
- Most POSIX System Calls

Be Consistent

- Stick to a metaphor for API names.
- Stick to a lexical naming scheme.
- Don’t mix plural and singular in names.
- Always use the same name for the same thing.
- Avoid off-by-one confusions.

Be Restrictive

- Make a clear distinction between internal and external APIs.
- Use accessor functions to access data structures.
- Don’t export struct internals when the user should only pass the pointer.
- Everything that does not affect the API can be changed easily!

Be Selective

- Limit the API to an easy to overview set of functions, if possible.
- Good example: PCRE API
- Provide an easy-to-use API for the most common use case, if feasible.
- Good example: CURL API

Free everything

- Provide `done()`, `destroy()` or `cleanup()` functions.
- Make sure that it is possible to free **every** resource allocated by your library.
- Use memory debuggers like `valgrind` to verify your implementation.

Bad example:
The QT library

Two-level-documentation

- Provide at least two levels of documentation:
  - The big picture / tutorial
  - Function, Class, etc. reference
- Tools like doxygen can only help with the latter one.
- Always assume that the reader of your documentation starts with zero knowlegde of your API.
- Always start with describing the bigger context.

Document the data layout

- Data structures etc. are almost ever under-documented.
- But often understanding the data structures is the key element for understanding an API.
- In good programs the data structure tends to dictate the imperative part of the program, not the other way around.

Bad example:
man pages

Stick to a freeing-paradigm

- Always be clear who is supposed to free what resource.
- Make a decision for one paradigm and stick to it.
- Freeing resources is the boring part.
- Nevertheless it's one of the most critical aspects in API design.

(Know your contracts!)

Program defensively

- Be liberal with what you acceppt.
- Be conservative with what you pass to the outside world.
- Check for the impossible case.
- Fail early when you have to fail.
- Leave all checks in the production code.

Support many languages

- Support as many programming languages as possible.
- Use the swig library to export a C/C++ API to the scripting world.
- Provide command line tools to make the API’s functionality accessible from an ordinary UNIX shell.
- Provide a thin C-wrapper for C++ libraries.
- Use MinGW or Cygwin to create a Win32 port of your library.

References

Books

- The Pragmatic Programmer
  Hunt and Thomas (ISBN-13: 978-0-201-61622-4)
- The Mythical Man-Month
  Brooks (ISBN-13: 978-0-201-83595-3)
- The Elements of Programming Style
  Kerninghan and Plauger (ISBN: 0-07-034207-5)

This Presentation

- Clifford Wolf
  `http://www.clifford.at/`
- More Presentations
  `http://www.clifford.at/papers/`
- This Presentation
  `http://www.clifford.at/papers/2008/apidesign/`